JAVA™ VIRTUAL MACHINE HARDWARE FOR RISC AND CISC PROCESSORS

BACKGROUND OF THE INVENTION

Java™ is an object orientated programming language developed by Sun Microsystems. The Java™ language is small, simple and portable across platforms and operating systems, both at the source and at the binary level. This makes the Java™ programming language very popular on the Internet.

Java™'s platform independence and code compaction are the most significant advantages of Java™ over conventional programming languages. In conventional programming languages, the source code of a program is sent to a compiler which translates the program into machine code or processor instructions. The processor instructions are native to the system's processor. If the code is compiled on an Intel-based system, the resulting program will only run on other Intel-based systems. If it is desired to run the program on another system, the user must go back to the original source code, obtain a compiler for the new processor, and recompile the program into the machine code specific to that other processor.

Java™ operates differently. The Java™ compiler takes a Java™ program and, instead of generating machine code for a particular processor, generates bytecodes. Bytecodes are instructions that look like machine code, but aren't specific to any processor. To execute a Java™ program, a bytecode interpreter takes the Java™ bytecode converts them to equivalent native processor instructions and executes the Java™ program. The Java™ byte code interpreter is one component of the Java™ Virtual Machine.

Having the Java™ programs in bytecode form means that instead of being specific to any one system, the programs can run on any platform and any operating

system as long a Java™ Virtual Machine is available. This allows a binary bytecode file to be executable across platforms.

The disadvantage of using bytecodes is execution speed. System specific programs that run directly on the hardware from which they are compiled, run significantly faster that Java™ bytecodes, which must be processed by the Java™ Virtual Machine. The processor must both convert the Java™ bytecodes into native instructions in the Java™ Virtual Machine and execute the native instructions.

One way to speed up the Java™ Virtual Machine is by techniques such as the "Just in Time" (JIT) interpreter, and even faster interpreters known as "Hot Spot JITs" interpreters. The JIT versions all result in a JIT compile overhead to generate native processor instructions. These JIT interpreters also result in additional memory overhead.

The slow execution speed of Java™ and overhead of JIT interpreters have made it difficult for consumer appliances requiring local-cost solutions with minimal memory usage and low energy consumption to run Java™ programs. The performance requirements for existing processors using the fastest JITs more than double to support running the Java™ Virtual Machine in software. The processor performance requirements could be met by employing superscalar processor architectures or by increasing the processor clock frequency. In both cases, the power requirements are dramatically increased. The memory bloat that results from JIT techniques, also goes against the consumer application requirements of low cost and low power.

It is desired to have an improved system for implementing Java™ programs that provides a low-cost solution for running Java™ programs for consumer appliances.

SUMMARY OF THE INVENTION

The present invention generally relates to a Java™ hardware accelerator which can be used to quickly translate Java™ bytecodes into native instructions for a central

processing unit (CPU). The hardware accelerator speeds up the processing of the Java™ bytecodes significantly because it removes the bottleneck which previously occurred when the Java™ Virtual Machine is run in software on the CPU to translate Java™ bytecodes into native instructions.

In the present invention, at least part of the Java™ Virtual Machine is implemented in hardware as the Java™ hardware accelerator. The Java™ hardware accelerator and the CPU can be put together on a single semiconductor chip to provide an embedded system appropriate for use with commercial appliances. Such an embedded system solution is less expensive than a powerful superscalar CPU and has a relatively low power consumption.

The hardware Java™ accelerator can convert the stack-based Java™ bytecodes into a register-based native instructions on a CPU. The hardware accelerators of the present invention are not limited for use with Java™ language and can be used with any stack-based language that is to be converted to register-based native instructions. Also, the present invention can be used with any language that uses instructions, such as bytecodes, which run on a virtual machine.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be further understood from the following description in conjunction with the drawings.

Figure 1 is a diagram of the system of the present invention including the hardware Java™ accelerator.

Figure 2 is a diagram illustrating the use of the hardware Java™ accelerator of the present invention.

Figure 3 is a diagram illustrating some the details of a Java™ hardware accelerator of one embodiment of the present invention.

Figure 4 is a diagram illustrating the details of one embodiment of a Java™ accelerator instruction translation in the system of the present invention.

Figure 5 is a diagram illustration the instruction translation operation of one embodiment of the present invention.

Figure 6 is a diagram illustrating the instruction translation system of one embodiment of the present invention using instruction level parallelism.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 is a diagram of the system 20 showing the use of a hardware Java™ accelerator 22 in conjunction with a central processing unit 26. The Java™ hardware accelerator 22 allows part of the Java™ Virtual Machine to be implemented in hardware. This hardware implementation speeds up the processing of the Java™ byte codes. In particular, in a preferred embodiment, the translation of the Java™ bytecodes into native processor instructions is at least partially done in the hardware Java™ accelerator 22. This translation has been part of a bottleneck in the Java™ Virtual Machine when implemented in software. In Figure 1, instructions from the instruction cache 24 or other memory is supplied to the hardware Java™ accelerator 22. If these instruction are Java™ bytecode, the hardware Java™ accelerator 22 can convert these bytecodes into native processor instruction which are supplied through the multiplexer 28 to the CPU. If a non-Java™ code is used, the hardware accelerator can be by-passed using the multiplexer 26.

The Java™ hardware accelerator can do, some or all of the following tasks:

1. Java™ bytecode decode;
2. identifying and encoding instruction level parallelism (ILP), wherever possible;
3. translating bytecodes to native instructions;
4. managing the Java™ stack on a register file associated with the CPU or as a separate stack;
5. generating exceptions on instructions on predetermined Java™ byte codes;
6. switching to native CPU operation when native CPU code is provided;
7. performing bounds checking on array instructions; and
8. managing the variables on the register file associated with the CPU.

In a preferred embodiment, the Java™ Virtual Machine functions of bytecode interpreter, Java™ register, and Java™ stack are implemented in the hardware Java™ accelerator. The garbage collection heap and constant pool area can be maintained in normal memory and accessed through normal memory referencing.

The major advantages of the Java™ hardware accelerator is to increase the speed in which the Java™ Virtual Machine operates, and allow existing native language legacy applications, software base, and development tools to be used. A dedicated microprocessor in which the Java™ bytecodes were the native instructions would not have accesss to those legacy applications.

Although the Java™ hardware accelerator is shown in Figure 1 as separate from the central processing unit, the Java™ hardware accelerator can be incorporated into a central processing unit. In that case, the central processing unit has a Java™ hardware accelerator subunit to translate Java™ bytecode into the native instructions operated on by the main portion of the CPU.

Figure 2 is a state machine diagram that shows the operation of one embodiment of the present invention. Block 32 is the power-on state. During power-on, the multiplexer 28 is set to bypass the Java™ hardware accelerator. In block 34, the native instruction boot-up sequence is run. Block 36 shows the system in the native mode executing native instructions and by-passing the Java™ hardware accelerator.

In block 38, the system switches to the Java™ hardware accelerator mode. In the Java™ hardware accelerator mode, Java™ bytecode is transferred to the Java™ hardware accelerator 22, converted into native instructions then sent to the CPU for operation.

The Java™ accelerator mode can produce exceptions at certain Java™ bytecodes. These bytecodes are not processed by the hardware accelerator 22 but are processed in the CPU 26. As shown in block 40, the system operates in the native mode but the Java™ Virtual Machine is implemented in the CPU which does the bytecode translation and handles the exception created in the Java™ accelerator mode.

The longer and more complicated bytecodes that are difficult to handle in hardware can be selected to produce the exceptions. Figure 7 is a table showing one possible list of bytecodes which can cause exceptions in a preferred embodiment.

Figure 3 is a diagram illustrating details of one embodiment of the Java™ hardware accelerator of the present invention. The Java™ hardware accelerator includes Java™ accelerator instruction translation hardware 42. The instruction translation Unit 42 is used to convert Java™ bytecodes to native instructions. One embodiment of the Java™ accelerator instruction translation hardware 42 is described in more detail below with respect to Figure 4. This instruction translation hardware 42 uses data stored in hardware Java™ registers 44. The hardware Java™ Registers store the Java™ Registers defined in the Java™ Virtual Machine. The Java™ Registers

contain the state of the Java™ Virtual Machine, affect its operation, and are updated after each bytecode is executed. The Java™ registers in the Java™ virtual machine include the PC, the program counter indicating what bytecode is being executed; Optop, a pointer to the top of the operand stack; Frame, a pointer to the execution environment of the current method; and Vars, a pointer to the first local variable available of the currently executing method. The virtual machine defines these registers to be a single 32-bit word wide. The Java™ registers are also stored in the Java™ stack which can be implemented as the hardware Java™ stack 50 or the Java™ stack can be stored into the CPU associated register file.

In a preferred embodiment, the hardware Java™ registers 44 can include additional registers for the use of the instruction translation hardware 42. These registers can include a register indicating a switch to native instructions and a register indicating the version number of the system.

The Java™ PC can be used to obtain bytecode instructions from the instruction cache 24. In one embodiment the Java™ PC is multiplexed with the normal program counter 54 of the central processing unit 26 in multiplexer 52. The normal PC 54 is not used during the operation of the Java™ hardware bytecode translation. In another embodiment, the normal program counter 54 is used as the Java™ program counter.

The Java™ registers are a part of the Java™ Virtual Machine and should not be confused with the general registers 46 or 48 which are operated upon by the central processing unit 26. In one embodiment, the system uses the traditional CPU register file 46 as well as a Java™ CPU register file 48. When native code is being operated upon the multiplexer 56 connects the conventional register file 46 to the execution logic 26c of the CPU 26. When the Java™ hardware accelerator is active, the Java™ CPU register file 48 substitutes for the conventional CPU register file 46. In another embodiment, the conventional CPU register file 46 is used.

As described below with respect to Figures 3 and 4, the Java™ CPU register file 48, or in an alternate embodiment the conventional CPU register file 46, can be used to store portions of the operand stack and some of the variables. In this way, the native register-based instructions from the Java™ accelerator instruction translator 42 can operate upon the operand stack and variable values stored in the Java™ CPU register file 48, or the values stored in the conventional CPU register file 46. Data can be written in and out of the Java™ CPU register file 48 from the data cache or other memory 58 through the overflow/underflow line 60 connected to the memory arbiter 62. The overflow/underflow transfer of data to and from the memory to can done concurrently with the CPU operation. Alternately, the overflow/underflow transfer can be done explicitly while the CPU is not operating. The overflow/underflow bus 60 can be implemented as a tri-state bus or as two separate buses to read data in and write data out of the register file when the Java™ stack overflows or underflows.

The register files for the CPU could alternately be implemented as a single register file with native instructions used to manipulate the loading of operand stack and variable values to and from memory. Alternately, multiple Java™ CPU register files could be used: one register file for variable values, another register file for the operand stack values, and another register file for the Java™ frame stack holding the method environment information.

The Java™ accelerator controller (co-processing unit) 64 can be used to control the hardware Java™ accelerator, read in and out from the hardware Java™ registers 44 and Java™ stack 50, and flush the Java™ accelerator instruction translation pipeline upon a "branch taken" signal from the CPU execute logic 26c.

The CPU 26 is divided into pipeline stages including the instruction fetch 26a, instruction decode 26b, execute logic 26c, memory access logic 26d, and writeback

logic 26e. The execute logic 26c executes the native instructions and thus can determine whether a branch instruction is taken and issue the "branch taken" signal.

Figure 4 illustrates an embodiment of a Java™ accelerator instruction translator which can be used with the present invention. The instruction buffer 70 stores the bytecode instructions from the instruction cache. The bytecodes are sent to a parallel decode unit 72 which decodes multiple bytecodes at the same time. Multiple bytecodes are processed concurrently in order to allow for instruction level parallelism. That is, multiple bytecodes may be converted into a lesser number of native instructions.

The decoded bytecodes are sent to a state machine unit 74 and Arithmetic Logic Unit (ALU) 76. The ALU 76 is provided to rearrange the bytecode instructions to make them easier to be operated on by the state machine 74. The state machine 74 converts the bytecodes into native instructions using the look-up table 78. Thus, the state machine 74 provides an address which indicates the location of the desired native instruction in the look-up table 78 . Counters are maintained to keep a count of how many entries have been placed on the operand stack, as well as to keep track of the top of the operand stack. In a preferred embodiment, the output of the look-up table 78 is augmented with indications of the registers to be operated on at line 80. The register indications are from the counters and interpreted from bytecodes. Alternately, these register indications can be sent directly to the Java™ CPU register file 48 shown in Figure 3.

The state machine 74 has access to the Java™ registers in 44 as well as an indication of the arrangement of the stack and variables in the Java™ CPU register file 48 or in the conventional CPU register file 46. The buffer 82 supplies the translated native instructions to the CPU.

The operation of the Java™ hardware accelerator of one embodiment of the present invention is illustrated in Figures 5 and 6. Figure 5, section I shows the instruction translation of the Java™ bytecode. The Java™ bytecode corresponding to the mnemonic iadd is interpreted by the Java™ virtual machine as an integer operation taking the top two values of the operand stack, adding them together and pushing the result on top of the operand stack. The Java™ translating machine translates the Java™ bytecode into a native instruction such as the instruction ADD R1, R2. This is an instruction native to the CPU indicating the adding of value in register R1 to the value in register R2 and the storing of this result in register R2 . R1 and R2 are the top two entries in the operand stack.

As shown in Figure 5, section II, the Java™ register includes a PC value of "Value A" that is incremented to "Value A+1". The Optop value changes from "Value B" to "Value B-1" to indicate that the top of the operand stack is at a new location. The Vars value which points to the top of the variable list is not modified. In Figure 5, section III, the contents of a Java™ CPU register file, such as the Java™ CPU register file 48 in Figure 3, is shown. The Java™ CPU register file starts off with registers R0-R5 containing operand stack values and registers R6-R7 containing variable values. Before the operation of the native instruction, register R1 contains the top value of the operand stack. Register R6 contains the first variable. After the execution of the native instruction, register R2 now contains the top value of the operand stack. Register R1 no longer contains a valid operand stack value and is available to be overwritten by a operand stack value from the memory sent across the overflow/underflow line 60 or from the bytecode stream.

Figure 5, section IV shows the memory locations of the operand stack and variables which can be stored in the data cache 58 or in main memory. For convenience, the memory is illustrated without illustrating any virtual memory

scheme. Before the native instruction executes, the address of the top of the operand stack, Optop, is "Value B". After the native instruction executes, the address of the top of the operand stack is "Value B-1" containing the result of the native instruction. Note that the operand stack value "4427" can be written into register R1 across the overflow/underflow line 60. Upon a switch back to the native mode, the data in the Java™ CPU register file 48 should be written to the data memory.

Consistency must be maintained between the Hardware Java™ Registers 44, the Java™ CPU register file 48 and the data memory. The CPU 26 and Java™ Accelerator Instruction Translation Unit 42 are pipelined and any changes to the hardware Java™ registers 44 and changes to the control information for the Java™ CPU register file 48 must be able to be undone upon a "branch taken" signal. The system preferably uses buffers (not shown) to ensure this consistency. Additionally, the Java™ instruction translation must be done so as to avoid pipeline hazards in the instruction translation unit and CPU.

Figure 6 is a diagram illustrating the operation of instruction level parallelism with the present invention. In Figure 6 the Java™ bytecodes iload_n and iadd are converted by the Java™ bytecode translator to the single native instruction ADD R6, R1. In the Java™ Virtual Machine, iload_n pushes the top local variable indicated by the by the Java™ register VAR onto the operand stack.

In the present invention the Java™ hardware translator can combine the iload_n and iadd bytecode into a single native instruction. As shown in figure 6, section II, the Java™ Register, PC, is updated from "Value A" to "Value A+2". The Optop value remains "value B". The value Var remains at "value C".

As shown in Figure 6, section III, after the native instruction ADD R6, R1 executes the value of the first local variable stored in register R6, "1221", is added to the value of the top of the operand stack contained in register R1 and the result stored

in register R1. In Figure 6, section IV, the Optop value does not change but the value in the top of the register contains the result of the ADD instruction, 1371.

The Java™ hardware accelerator of the present invention is particularly well suited to a embedded solution in which the hardware accelerator is positioned on the same chip as the existing CPU design. This allows the prior existing software base and development tools for legacy applications to be used. In addition, the architecture of the present embodiment is scalable to fit a variety of applications ranging from smart cards to desktop solutions. This scalability is implemented in the Java™ accelerator instruction translation unit of Figure 4. For example, the lookup table 78 and state machine 74 can be modified for a variety of different CPU architectures. These CPU architectures include reduced instruction set computer (RISC) architectures as well as complex instruction set computer (CISC) architectures. The present invention can also be used with superscalar CPUs or very long instruction word (VLIW) computers.

While the present invention has been described with reference to the above embodiments, this description of the preferred embodiments and methods is not meant to be construed in a limiting sense. For example, the term Java™ in the specification or claims should be construed to cover successor programming languages or other programming languages using basic Java™ concepts (the use of generic instructions, such as bytecodes, to indicate the operation of a virtual machine). It should also be understood that all aspects of the present invention are not to be limited to the specific descriptions, or to configurations set forth herein. Some modifications in form and detail the various embodiments of the disclosed invention, as well as other variations in the present invention, will be apparent to a person skilled in the art upon reference to the present disclosure. It is therefore contemplated that the following claims will

cover any such modifications or variations of the described embodiment as falling within the true spirit and scope of the present invention.